20 | 土地需求扩大与保障：如何表示虚拟内存？

2021-06-23 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 12:19 大小 11.30M

你好，我是 LMOS。

在现实中，有的人需要向政府申请一大块区域，在这块区域中建楼办厂，但是土地有限且已经被占用。所以可能的方案是，只给你分配一个总的面积区域，今年湖北有空地就在湖北建立一部分厂房，明年广东有空地就在广东再建另一部分厂房，但是总面积不变。

其实在计算机系统中也有类似的情况，一个应用往往拥有很大的连续地址空间，并且每个应用都是一样的，只有在运行时才能分配到真正的物理内存，在操作系统中这称为虚拟内存。

那问题来了，操作系统要怎样实现虚拟内存呢？由于内容比较多，我会用两节课的时间带你解决这个问题。今天这节课，我们先进行虚拟地址空间的划分，搞定虚拟内存数据结构

的设计。下节课再动手实现虚拟内存的核心功能。

好，让我们进入正题，先从虚拟地址空间的划分入手，配套代码你可以从这里获得。

虚拟地址空间的划分

虚拟地址就是逻辑上的一个数值，而虚拟地址空间就是一堆数值的集合。通常情况下，32 位的处理器有 0～0xFFFFFFFF 的虚拟地址空间，而 64 位的虚拟地址空间则更大，有 0～0xFFFFFFFFFFFFFFFF 的虚拟地址空间。

对于如此巨大的地址空间，我们自然需要一定的安排和设计，**比如什么虚拟地址段放应用，什么虚拟地址段放内核等。**下面我们首先看看处理器硬件层面的划分，再来看看在此基础上我们系统软件层面是如何划分的。

x86 CPU 如何划分虚拟地址空间

我们 Cosmos 工作在 x86 CPU 上，所以我们先来看看 x86 CPU 是如何划分虚拟地址空间的。

由于 x86 CPU 支持虚拟地址空间时，要么开启保护模式，要么开启长模式，保护模式下是 32 位的，有 0～0xFFFFFFFF 个地址，可以使用完整的 4GB 虚拟地址空间。

在保护模式下，对这 4GB 的虚拟地址空间没有进行任何划分，而长模式下是 64 位的虚拟地址空间有 0～0xFFFFFFFFFFFFFFFF 个地址，这个地址空间非常巨大，硬件工程师根据需求设计，把它分成了 3 段，如下图所示。

x86虚拟地址划分

长模式下，CPU 目前只实现了 48 位地址空间，但寄存器却是 64 位的，CPU 自己用地址数据的第 47 位的值扩展到最高 16 位，所以 64 位地址数据的最高 16 位，要么是全 0，要么全 1，这就是我们在上图看到的情形。

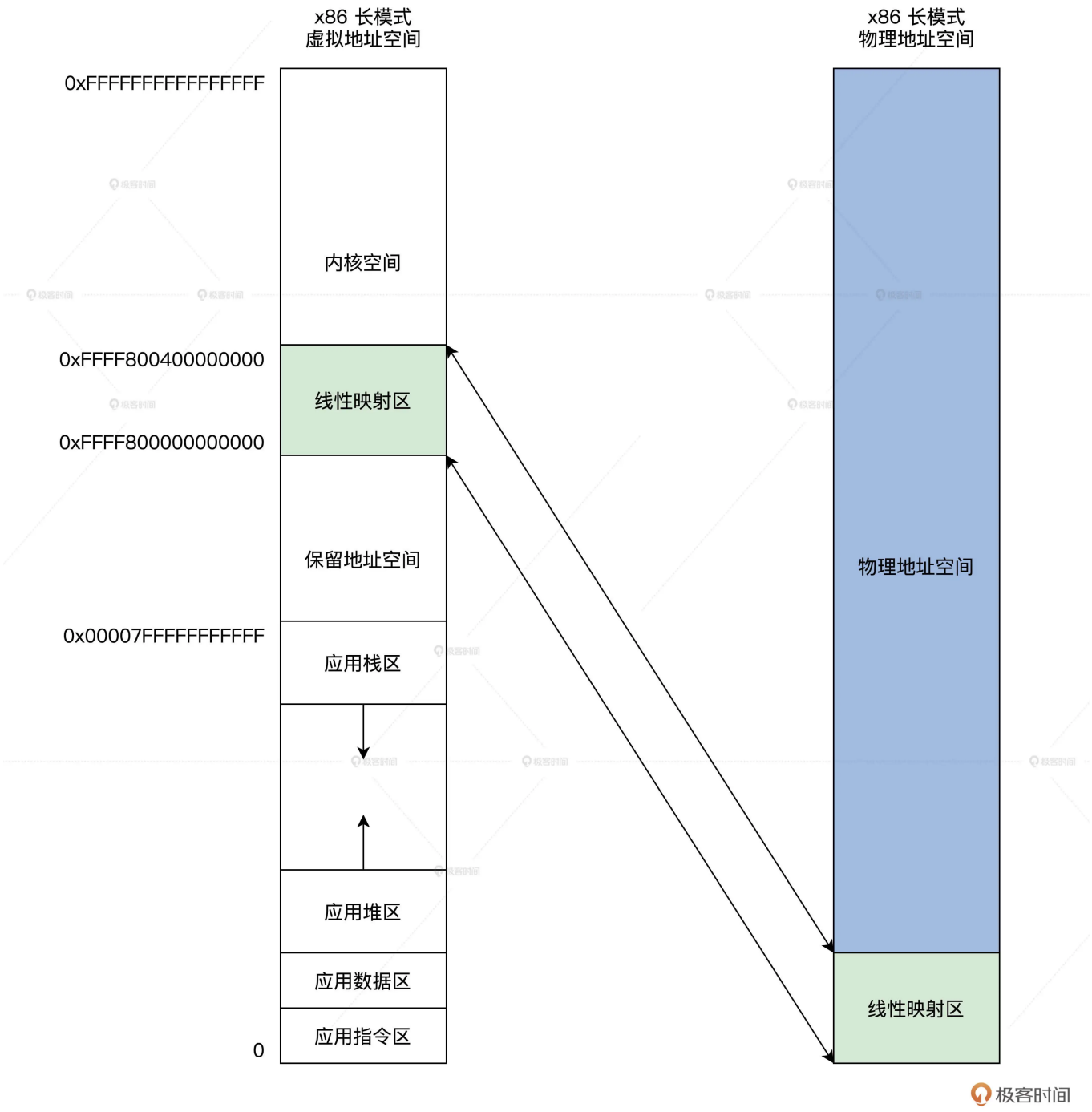
Cosmos 如何划分虚拟地址空间

现在我们来规划一下，Cosmos 对 x86 CPU 长模式下虚拟地址空间的使用。由前面的图形可以看出，在长模式下，整个虚拟地址空间只有两段是可以用的，很自然一段给内核，另一段就给应用。

我们把 0xFFFF800000000000～0xFFFFFFFFFFFFFFFF 虚拟地址空间分给内核，把 0～0x00007FFFFFFFFFFF 虚拟地址空间分给应用，内核占用的称为内核空间，应用占用的就叫应用空间。

在内核空间和应用空间中，我们又继续做了细分。后面的图并不是严格按比例画的，应用程序在链接时，会将各个模块的指令和数据分别放在一起，应用程序的栈是在最顶端，向下增长，应用程序的堆是在应用程序数据区的后面，向上增长。

内核空间中有个线性映射区 0xFFFF800000000000～0xFFFF800400000000，这是我们在二级引导器中建立的 **MMU 页表映射**。



内核空间与应用空间

如何设计数据结构

根据前面经验，我们要实现一个功能模块，首先要设计出相应的数据结构，虚拟内存模块也一样。

这里涉及到虚拟地址区间，管理虚拟地址区间以及它所对应的物理页面，最后让进程和虚拟地址空间相结合。这些数据结构小而多，下面我们一个个来设计。

虚拟地址区间

我们先来设计虚拟地址区间数据结构，由于虚拟地址空间非常巨大，我们绝不能像管理物理内存页面那样，一个页面对应一个结构体。那样的话，我们整个物理内存空间或许都放不下所有的虚拟地址区间数据结构的实例变量。

由于虚拟地址空间往往是以区为单位的，比如栈区、堆区，指令区、数据区，这些区内部往往是连续的，区与区之间却间隔了很大空间，而且每个区的空间扩大时我们不会建立新的虚拟地址区间数据结构，而是改变**其中的指针**，这就节约了内存空间。

下面我们来设计这个数据结构，代码如下所示。

复制代码

```
typedef struct KMVARSDSC
{
    spinlock_t kva_lock;        //保护自身自旋锁
    u32_t  kva_maptype;         //映射类型
    list_h_t kva_list;          //链表
    u64_t  kva_flgs;            //相关标志
    u64_t  kva_limits;
    void*  kva_mcstruct;        //指向它的上层结构
    adr_t  kva_start;           //虚拟地址的开始
    adr_t  kva_end;             //虚拟地址的结束
    kvmemcbox_t* kva_kvmbox;    //管理这个结构映射的物理页面
    void*  kva_kvmcobj;
}kmvarsdsc_t;
```

如你所见，除了自旋锁、链表、类型等字段外，最重要的就是**虚拟地址的开始与结束字段**，它精确描述了一段虚拟地址空间。

整个虚拟地址空间如何描述

有了虚拟地址区间的数据结构，怎么描述整个虚拟地址空间呢？我们整个的虚拟地址空间，正是由多个虚拟地址区间连接起来组成，也就是说，只要把许多个虚拟地址区间数据结构按顺序连接起来，就可以表示整个虚拟地址空间了。

这个数据结构我们这样来设计。

复制代码

```
typedef struct s_VIRMEMADRS
{
    spinlock_t vs_lock;             //保护自身的自旋锁
    u32_t  vs_resalin;
    list_h_t vs_list;               //链表，链接虚拟地址区间
    uint_t vs_flgs;                 //标志
    uint_t vs_kmvdscnr;             //多少个虚拟地址区间
    mmadrsdsc_t* vs_mm;             //指向它的上层的数据结构
    kmvarsdsc_t* vs_startkmvdsc;    //开始的虚拟地址区间
    kmvarsdsc_t* vs_endkmvdsc;      //结束的虚拟地址区间
    kmvarsdsc_t* vs_currkmvdsc;     //当前的虚拟地址区间
    adr_t vs_isalcstart;            //能分配的开始虚拟地址
    adr_t vs_isalcend;              //能分配的结束虚拟地址
    void* vs_privte;                //私有数据指针
    void* vs_ext;                   //扩展数据指针
}virmemadrs_t;
```

从上述代码可以看出，virmemadrs_t 结构管理了整个虚拟地址空间的 kmvarsdsc_t 结构，kmvarsdsc_t 结构表示一个虚拟地址区间。这样我们就能知道，虚拟地址空间中哪些地址区间没有分配，哪些地址区间已经分配了。

进程的内存地址空间

虚拟地址空间作用于应用程序，而应用程序在操作系统中用进程表示。

当然，一个进程有了虚拟地址空间信息还不够，还要知道进程和虚拟地址到物理地址的映射信息，应用程序文件中的指令区、数据区的开始、结束地址信息。

所以，我们要把这些信息综合起来，才能表示一个进程的完整地址空间。这个数据结构我们可以这样设计，代码如下所示。

复制代码

```
typedef struct s_MMADRSDSC
{
    spinlock_t msd_lock;                //保护自身的自旋锁
    list_h_t msd_list;                  //链表
    uint_t msd_flag;                    //状态和标志
    uint_t msd_stus;
    uint_t msd_scount;                  //计数，该结构可能被共享
```

```
    sem_t  msd_sem;                    //信号量
    mmudsc_t msd_mmu;                  //MMU相关的信息
    virmemadrs_t msd_virmemadrs;       //虚拟地址空间
    adr_t msd_stext;                   //应用的指令区的开始、结束地址
    adr_t msd_etext;
    adr_t msd_sdata;                   //应用的数据区的开始、结束地址
    adr_t msd_edata;
    adr_t msd_sbss;
    adr_t msd_ebss;
    adr_t msd_sbrk;                    //应用的堆区的开始、结束地址
    adr_t msd_ebrk;
}mmadrsdsc_t;
```

进程的物理地址空间，其实可以用一组 MMU 的页表数据表示，它保存在 mmudsc_t 数据结构中，但是这个数据结构我们不在这里研究，放在后面再研究。

页面盒子

我们知道每段虚拟地址区间，在用到的时候都会映射对应的物理页面。根据前面我们物理内存管理器的设计，每分配一个或者一组内存页面，都会返回一个 msadsc_t 结构，所以我们还需要一个数据结构来挂载 msadsc_t 结构。

但为什么不直接挂载到 kmvarsdsc_t 结构中去，而是要设计一个新的数据结构呢？

我们当然有自己的考虑，一般虚拟地址区间是和文件对应的数据相关联的。比如进程的应用程序文件，又比如把一个文件映射到进程的虚拟地址空间中，只需要在内存页面中保留一份共享文件，多个程序就都可以共享它。

常规操作就是把同一个物理内存页面映射到不同的虚拟地址区间，所以我们实现一个专用的数据结构，共享操作时就可以让多个 kmvarsdsc_t 结构指向它，代码如下所示。

复制代码

```
typedef struct KVMEMCBOX
{
    list_h_t kmb_list;         //链表
    spinlock_t kmb_lock;       //保护自身的自旋锁
    refcount_t kmb_cont;       //共享的计数器
    u64_t kmb_flgs;            //状态和标志
    u64_t kmb_stus;
    u64_t kmb_type;            //类型
    uint_t kmb_msanr;          //多少个msadsc_t
```

```
    list_h_t kmb_msalist;      //挂载msadsc_t结构的链表
    kvmemcboxmgr_t* kmb_mgr;   //指向上层结构
    void* kmb_filenode;        //指向文件节点描述符
    void* kmb_pager;           //指向分页器 暂时不使用
    void* kmb_ext;             //自身扩展数据指针
}kvmemcbox_t;
```

到这里为止，一个内存页面容器盒子就设计好了，它可以独立存在，又和虚拟内存区间有紧密的联系，甚至可以用来管理文件数据占用的物理内存页面。

页面盒子的头

kvmemcbox_t 结构是一个独立的存在，我们必须能找到它，所以还需要设计一个全局的数据结构，用于管理所有的 kvmemcbox_t 结构。这个结构用于挂载 kvmemcbox_t 结构，对其进行计数，还要支持缓存多个空闲的 kvmemcbox_t 结构，代码如下所示。

复制代码

```
typedef struct KVMEMCBOXMGR
{
    list_h_t kbm_list;         //链表
    spinlock_t kbm_lock;       //保护自身的自旋锁
    u64_t kbm_flgs;            //标志与状态
    u64_t kbm_stus;
    uint_t kbm_kmbnr;          //kvmemcbox_t结构个数
    list_h_t kbm_kmbhead;      //挂载kvmemcbox_t结构的链表
    uint_t kbm_cachenr;        //缓存空闲kvmemcbox_t结构的个数
    uint_t kbm_cachemax;       //最大缓存个数，超过了就要释放
    uint_t kbm_cachemin;       //最小缓存个数
    list_h_t kbm_cachehead;    //缓存kvmemcbox_t结构的链表
    void* kbm_ext;             //扩展数据指针
}kvmemcboxmgr_t;
```

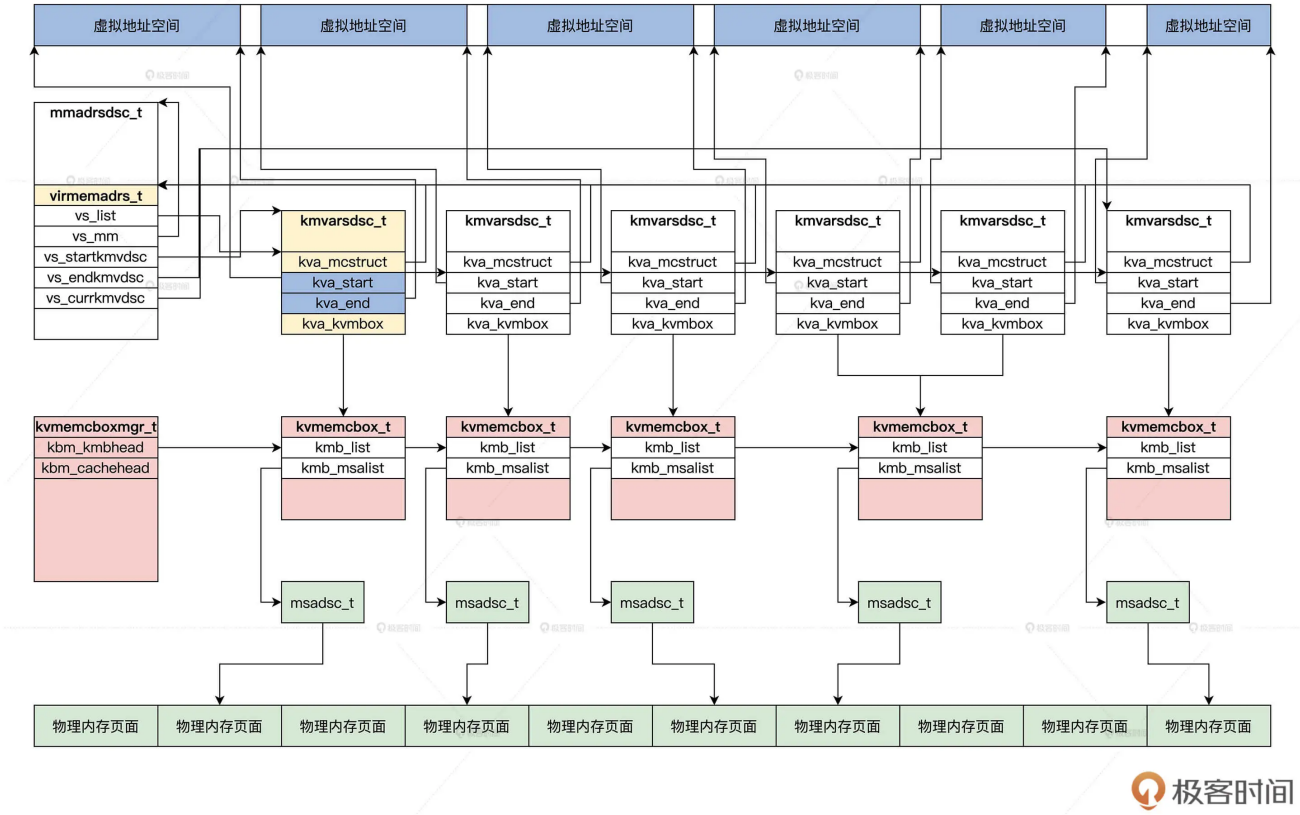
上述代码中的缓存相关的字段，是为了防止频繁分配、释放 kvmemcbox_t 结构带来的系统性能抖动。同时，缓存几十个 kvmemcbox_t 结构下次可以取出即用，不必再找内核申请，这样可以大大提高性能。

理清数据结构之间的关系

现在，所有的数据结构已经设计完成，比较多。其中每个数据结构的功能我们已经清楚了，唯一欠缺的是，我们还没有明白它们之间的关系是什么。

只有理清了它们之间的关系，你才能真正明白，它们组合在一起是怎么完成整个功能的。

我们在写代码时，脑中有图，心中才有底。这里我给你画了一张图，为了降低复杂性，我并没有画出数据结构的每个字段，图里只是表达一下它们之间的关系。



虚拟内存数据结构

这张图你需要按照从上往下、从左到右来看。首先从进程的虚拟地址空间开始，而进程的虚拟地址是由 kmvarsdsc_t 结构表示的，一个 kmvarsdsc_t 结构就表示一个已经分配出去的虚拟地址空间。一个进程所有的 kmvarsdsc_t 结构，要交给进程的 mmadrsdsc_t 结构中的 virmemadrs_t 结构管理。

我们继续往下看，为了管理虚拟地址空间对应的物理内存页面，我们建立了 kvmembox_t 结构，它由 kvmemcboxmgr_t 结构统一管理。在 kvmembox_t 结构中，挂载了物理内存页面对应的 msadsc_t 结构。

整张图片完整地展示了从虚拟内存到物理内存的关系，理清了这些数据结构关系之后，我们就可以写代码实现了。

初始化

由于我们还没有讲到进程相关的章节，而虚拟地址空间的分配与释放，依赖于进程数据结构下的 mmadrsdsc_t 数据结构，所以我们得想办法产生一个 mmadrsdsc_t 数据结构的实例变量，最后初始化它。

下面我们先在 cosmos/kernel/krlglobal.c 文件中，申明一个 mmadrsdsc_t 数据结构的实例变量，代码如下所示。

复制代码

```
KRL_DEFGLOB_VARIABLE(mmadrsdsc_t, initmmadrsdsc);
```

接下来，我们要初始化这个申明的变量，操作也不难。因为这是属于内核层的功能了，所以要在 cosmos/kernel/ 目录下建立一个模块文件 krlvadrsmem.c，在其中写代码，如下所示。

复制代码

```
bool_t kvma_inituserspace_virmemadrs(virmemadrs_t *vma)
{
    kmvarsdsc_t *kmvdc = NULL, *stackkmvdc = NULL;
    //分配一个kmvarsdsc_t
    kmvdc = new_kmvarsdsc();
    if (NULL == kmvdc)
    {
        return FALSE;
    }
    //分配一个栈区的kmvarsdsc_t
    stackkmvdc = new_kmvarsdsc();
    if (NULL == stackkmvdc)
    {
        del_kmvarsdsc(kmvdc);
        return FALSE;
    }
    //虚拟区间开始地址0x1000
    kmvdc->kva_start = USER_VIRTUAL_ADDRESS_START + 0x1000;
    //虚拟区间结束地址0x5000
    kmvdc->kva_end = kmvdc->kva_start + 0x4000;
    kmvdc->kva_mcstruct = vma;
    //栈虚拟区间开始地址0x1000USER_VIRTUAL_ADDRESS_END - 0x40000000
    stackkmvdc->kva_start = PAGE_ALIGN(USER_VIRTUAL_ADDRESS_END - 0x40000000);
    //栈虚拟区间结束地址0x1000USER_VIRTUAL_ADDRESS_END
    stackkmvdc->kva_end = USER_VIRTUAL_ADDRESS_END;
    stackkmvdc->kva_mcstruct = vma;

    knl_spinlock(&vma->vs_lock);
```

```
    vma->vs_isalcstart = USER_VIRTUAL_ADDRESS_START;
    vma->vs_isalcend = USER_VIRTUAL_ADDRESS_END;
    //设置虚拟地址空间的开始区间为kmvdc
    vma->vs_startkmvdsc = kmvdc;
    //设置虚拟地址空间的开始区间为栈区
    vma->vs_endkmvdsc = stackkmvdc;
    //加入链表
    list_add_tail(&kmvdc->kva_list, &vma->vs_list);
    list_add_tail(&stackkmvdc->kva_list, &vma->vs_list);
    //计数加2
    vma->vs_kmvdscnr += 2;
    knl_spinunlock(&vma->vs_lock);
    return TRUE;
}

void init_kvirmemadrs()
{
    //初始化mmadrsdsc_t结构非常简单
    mmadrsdsc_t_init(&initmmadrsdsc);
    //初始化进程的用户空间
    kvma_inituserspace_virmemadrs(&initmmadrsdsc.msd_virmemadrs);
}
```

上述代码中，init_kvirmemadrs 函数首先调用了 mmadrsdsc_t_init，对我们申明的变量进行了初始化。因为这个变量中有链表、自旋锁、信号量这些数据结构，必须要初始化才能使用。

最后调用了 kvma_inituserspace_virmemadrs 函数，这个函数中建立了一个虚拟地址区间和一个栈区，栈区位于虚拟地址空间的顶端。下面我们在 krlinit..c 中的 init_krl 函数中来调用它。

复制代码

```
void init_krl()
{
    //初始化内核功能层的内存管理
    init_krlmm();
    die(0);
    return;
}
void init_krlmm()
{
    init_kvirmemadrs();
    return;
}
```

至此，我们的内核功能层的初始流程就建立起来了，是不是很简单呢？

重点回顾

至此我们关于虚拟内存的虚拟地址空间的划分和虚拟内存数据结构的设计就结束了，我把这节课的重点为你梳理一下。

首先是虚拟地址空间的划分。由于硬件平台的物理特性，虚拟地址空间被分成了两段，Cosmos 也延续了这种划分的形式，顶端的虚拟地址空间为内核占用，底端为应用占用。内核还建立了 16GB 的线性映射区，而应用的虚拟地址空间分成了指令区，数据区，堆区，栈区。

然后为了实现虚拟地址内存，我们设计了大量的数据结构，它们分别是虚拟地址区间 kmvarsdsc_t 结构、管理虚拟地址区间的虚拟地址空间 virmemadrs_t 结构、包含 virmemadrs_t 结构和 mmudsc_t 结构的 mmadrsdsc_t 结构、用于挂载 msadsc_t 结构的页面盒子的 kvmemcbox_t 结构、还有用于管理所有的 kvmemcbox_t 结构的 kvmemcboxmgr_t 结构。

结构名称	功能及描述
kmvarsdsc_t 结构	虚拟地址区间
virmemadrs_t 结构	管理整个虚拟地址空间的 kmvarsdsc_t 结构
mmadrsdsc_t 结构	进程数据结构下的一个结构（进程部分还会细讲），它包含 virmemadrs_t 结构和 mmudsc_t 结构
kvmemcbox_t 结构	用于挂载msadsc_t结构的页面盒子
kvmemcboxmgr_t 结构	用于管理所有的 kvmemcbox_t 结构

数据结构功能表

最后是初始化工作。由于我们还没有进入到进程相关的章节，所以这里必须要申明一个进程相关的 mmadrsdsc_t 结构的实例变量，并进行初始化，这样我们才能测试虚拟内存的功能。

思考题

请问内核虚拟地址空间为什么有一个 0xFFFF800000000000 ~ 0xFFFF800400000000 的线性映射区呢？

欢迎你在留言区跟我交流讨论。如果这节课对你有帮助，也欢迎你分享给你的朋友。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 土地不能浪费：如何管理内存对象？

下一篇 21 | 土地需求扩大与保障：如何分配和释放虚拟内存？

精选留言 (6)

写留言

二三子也
2021-06-23

内核代码使用虚拟地址，但是内核有时需要用到物理地址，比如设置页表项等。线性映射区使得内核能通过加减一个固定值的方式，方便的完成虚拟地址与物理地址的转换。

展开

作者回复: 你好 ， 正确的

4

pedro
2021-06-23

0xFFFF800000000000～0xFFFF800400000000 的线性映射区是MMU 页表映射数据，保存虚拟地址和物理地址之间的映射关系，没有这块区域，CPU无法在长模式下工作，通过虚拟地址将无法访问真实的数据。

且这个区域必须在内核态，放在用户态太危险了，万一被某个应用程序修改，直接爆炸。

展开

作者回复: 也是正确的，大写6666

1

青玉白露
2021-07-13

这个线性映射区主要是用来存放MMU的
试想，如果所有的地址都是物理地址转换到虚拟地址，这个转换关系存在哪？存在虚拟内存里？那这不是套娃么，你连MMU都找不到，根本没法转换。
所以才有了一段映射区，内核态直接去访问这个区域就可以获取MMU的数据，避免套娃。

展开

作者回复: 是的 哈哈

Feen
2021-07-10

最后的思考题：因为不管是实模式还是保护模式或者长模式，物理内存 0～0x400000000 的实体空间是被系统的启动类型文件占用，这里不止包括内核相关的，还有BIOS中断表等等前面课程讲过的硬软件系统必须要的文件，开机就占用，关机谁也占用不了，首先给这段物理地址就固定下来。这是必要的。
而操作系统运行在非实模式下虚拟地址空间之后，如果不建立线性映射表，那对这块物...
展开

作者回复: 嗯 嗯

相逢是缘
2021-06-23

对于思考题自己的理解：
因为开启长模式和MMU之后，访问的所有地址都应该是“虚拟地址”，这个地址必须经过MMU转换才能访问到真正的物理地址。而我们的内核代码以及数据都放在物理地址的0x0~0x400000000内，当划分0xffff800000000000为内核虚拟地址的开始地址时，需要...
展开

作者回复: 你说的对

blentle
2021-06-23

是不是因为内核地址空间是共享的，而内核页表部分是又是相同的，才会有一个线性映射目的是相对于外部程序来说仿佛是访问不同的内存空间.不知道理解的对不对……真的是越来越难，上一节还没消化

作者回复: 是啊